APPLICATION

FOR

UNITED STATES LETTERS PATENT

09595-004002 11/14/00 15:52:00

TITLE: SOFTWARE VAULT

APPLICANT: CADIR BATISTA LEE AND SCOTT WILLIAM DALE

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL557831914US

November 14, 2000

Date of Deposit

Ins. B1

SOFTWARE VAULT

Background

Inexorable advances in electronics have led to widespread deployment of inexpensive, yet powerful computers that are networked together. Over time, programs installed on these computers are updated and these updates need to be maintained. Information system departments and their users face the thorny task of maintaining numerous instances of software across their complex distributed environments. The maintenance process covers a number of tasks, including software installation, synchronization, backup, recovery, analysis and repair.

A detailed knowledge of a computer's dynamic environment and its system configuration is needed in the maintenance process to prevent situations where modifications to one component may introduce problems in other components. Moreover, an accurate knowledge of the system's configuration is required to verify compatibility and to ensure integrity across multiple operating environments and across diverse processors. Software applications can have numerous components and dependencies and, in a modern network with diverse processors and peripherals, the range of possible configurations can be staggering.

Historically, relationships between software components have been manually detected and component states have been recorded in a log. This state information is external of the components themselves and must be updated whenever the components change. As state information is recorded only at the time of installation, changes made subsequent to installation may be lost. As the rate of change increases and complexity of the software configuration grows, the external state representation becomes difficult to maintain and prone to error. Moreover, during normal operation, users may make changes to the software through individual personalization and through the installation

of additional software, thus changing the state information. The difference in state information between software installation and software operation can lead to unpredictable behavior and may require support from information system personnel.

Summary of the Invention

In one aspect, a computer-implemented vault archives software components, where only a single instance of each component that is multiply-used is stored in the vault. The vault includes unique instances of the one or more software components and an access controller for performing a direct, random access retrieval of the one or more software components from the vault.

Implementations of the invention include one or more of the following. The access controller generates a unique key. The unique key may be used to access a software component. The unique key may be generated from a persistent metadata description. A post controller may perform a direct, random access insertion of a software component to the vault. The post controller may generate a unique key from the new component and optimizes storage if the unique key exists. A look-up controller may perform a direct, random access determination of the existence of a software component in the vault. A client may be coupled to the vault, the client having a physical software component residing on the client, the client generating a key from the physical software component. One or more secondary vaults may be coupled to the vault with a fault-tolerant rollover system for sequentially searching each vault for the presence of a target software component. The secondary vaults may be ordered based on accessibility of the vaults. A client may generate a key and apply the key to recover the target software component from the most accessible of the vaults. The client may use a metadata description to generate the key. If the search of a determined vault fails to locate the target software component, the client may skip the determined vault and modify the search

order of the vaults in recovering the target software component.

In a second aspect, a computer-implemented vault archives software components, where only a single instance of each component that is multiply-used is stored in the vault. The vault includes means for storing unique instances of the one or more software components on the vault; and access means for performing a direct, random access retrieval of the one or more software components from the vault.

Implementations of the invention include one or more of the following. The access means may generate a unique key. The unique key may be used to access a software component. The unique key may be generated from a persistent metadata description. A post means may perform a direct, random access insertion of a software component to the vault. The post means may generate a unique key from the new component and optimize storage if the unique key exists. A look-up means may perform a direct, random access determination of the existence of a software component in the vault. A client may be coupled to the vault, the client having a physical software component residing on the client, the client generating a key from the physical software component. One or more secondary vaults may be coupled to the vault with means for sequentially searching each vault for the presence of a target software component. The secondary vaults may be ordered based on accessibility of the vaults. The client may have a means for applying the key to recover the target software component from the most accessible of the vaults. The client may use a metadata description to generate the key. If the search of a determined vault fails to locate the target software component, the client may skip the determined vault and modifies the search order of the vaults in recovering the target software component.

In a third aspect, a method for archiving software components where only a single instance of each component that is multiply-used is stored in a vault includes: storing unique instances of the one or more software components in the vault;

and performing a direct, random access retrieval of the one or more software components from the vault.

Implementations of the invention include one or more of the following. The method may generate a unique key. The unique key may be used to access a software component. The unique key may be generated from a persistent metadata description. The method may perform a direct, random access insertion of a software component to the vault. A unique key may be generated from the new component and used to optimize storage if the unique key exists. The method may perform a direct, random access determination of the existence of a software component in the vault. A client may be coupled to the vault, the client having a physical software component residing on the client, the client generating a key from the physical software component. One or more secondary vaults may be coupled to the vault and sequentially searched for the presence of a target software component. The secondary vaults may be ordered based on accessibility of the vaults. The client may apply the key to recover the target software component from the most accessible of the vaults. The client may use a metadata description to generate the key. If the search of a determined vault fails to locate the target software component, the client may skip the determined vault and modifies the search order of the vaults in recovering the target software component.

Advantages of the invention include one or more of the following. The vault can inventory, install, deploy, maintain, repair and optimize software across local and wide area networks (LANs and WANs). By automating the human-intensive process of managing software throughout its life cycle, the vault reduces the total cost of ownership of computers in networked environments. Users can reduce the time required for software packaging by simply probing an application for its current state and storing unique instances of components of the software in a vault. Software installation may then be performed by moving valid working states from one client machine to another.

Further, error prone installation of the software is avoided, increasing the out-of-box success by installing known working software states and insuring against deletion of shared components.

Moreover, the vault can be used to diagnose problems by comparing an existing state on a client computer to both a previously working state and a reference state stored in the vault. Further, the vault can be used to allow applications which have been damaged to self-heal applications by automatically restoring previously working states or reinstalling components from reference states.

The vault can also support remote and disconnected users by protecting applications on their desktop and ensuring that software is configured appropriately. The vault can also synchronize user desktops by automatically updating all application components and configuration settings while still allowing custom settings for the user. The vault also automates custom computer setups/upgrades by providing replication of working states from client machines. Information stored in the vault may be used to provide vital application information including system values and resource conflicts to help information systems personnel.

Further, the vault decreases network overhead and increases scalability of electronic software distribution by eliminating delivery of duplicate files that make up software packages. The flexible architecture of the invention protects user investment in existing solutions for enterprise-wide systems management, network management, and application management.

Brief Description of the Drawings

Fig. 1 is a diagram illustrating a system with one or more vaults for communicating with one or more clients.

Fig. 2 is a diagram illustrating communications between the client and the one or more vaults.

Fig. 3 is a flowchart illustrating a process for accessing

software components from the vault.

Fig. 4 is a process illustrating a process for comparing keys in Fig. 3.

Fig. 5 is a flowchart illustrating a process for posting software components to the vault.

Fig. 6 is a flowchart illustrating in more detail the process for generating a post key in Fig. 5.

Fig. 7 is a flowchart illustrating a process for performing lookup based on an identity key.

Figs. 8A and 8B are flowcharts illustrating alternate processes for software management.

Fig. 9 is a flowchart illustrating a process for publishing meta data information associated with the software components.

Fig. 10 is a flowchart illustrating a process for storing software components on the vault.

Fig. 11 is a flowchart illustrating a process for replicating software components from the vault.

Fig. 12 is a flowchart illustrating an exemplary application of the vault in installing software.

Fig. 13 is a diagram of an exemplary application for maintaining software using the vault.

Description

Fig. 1 shows a computer system 100 with one or more vaults is shown. The system 100 has one or more clients 102, each of which has a set of catalogs 104, as well as a client vault 106. The client vault 106 is a computer readable medium which stores one or more software components (entities) which are designated by the set of catalogs 104. In this case, the client vault 106 exists on a data storage device such as a hard drive on the computer system 100. Alternatively, the vault may reside on one or more data storage devices connected to a network 110, as discussed below. Each of the software components may be referenced by more than one application, and the software

components are used to reconstruct the application.

Each catalog 104 includes metadata which is generated by determining run-time states of each software application. Generally, the metadata for each software application is an abstract representation of a predetermined set of functionalities tailored for a particular user during installation or customized during operation of the software application. The metadata is a list pointing to various software components (entities) making up an application software and a root entity which represents an action that may be performed by the user at another machine, place or time.

The metadata is generated by analyzing the run-time states of the software application and checking the existence of the entities and entity dependencies, which may change over time. The list of the entities is pruned by deleting overlapping entities and by combining similar entities. In the grouping of entities in the metadata, an intersection of the entities is determined such that a package of entities can be succinctly defined and that all the information necessary for it can be represented as the metadata with or without the actual entities. Enough information about each entity is included in the metadata so that an analysis of correctness may be performed. The resulting metadata provides indices to the client vault 106, which stores unique instances of software components locally to the client 102.

In addition to the client vault 106, the client 102 may also access remotely stored component files associated with the catalog 104. To access these remote component files, the client 102 communicates over the network 110, which may be an intranet, or a wide area network (WAN) such as the Internet. The network 110 may also be a local area network (LAN). Connected to the network 110 are one or more vaults 112, 114 and 116. Each of the vaults 112-116 includes sets of catalogs 118-120 which are indices of metadata files that represent the physical components of the software being "published" for the client 102.

Figure 2 shows in more detail various communication modules between a client 122 and one or more vaults 130-132. The vaults 130-132 may be local vaults, remote vaults accessible from a network, or a combination of both. In Figure 2, an access controller 124 allows the client 122 to retrieve files from the one or more vaults 130-132. A post controller 126 allows the client 122 to place one or more files on the vaults 130-132. A lookup controller 128 allows the client 122 to preview and compare catalogs of files stored locally versus files stored on the vaults 130-132. Each of controllers 124, 126 and 128 may be implemented using a processor on the client 122 and computer instructions as described below. Alternatively, each of controllers 124, 126 and 128 may be implemented using a processor which is located on the network 110. Pseudo-code showing file accesses using the access controller 124, the post controller 126 and the lookup controller 128 is as follows:

```
//**** Access Controller
Pass metadata descriptor
Transform metadata descriptor into key
for each vault
     directly access key on vault
     if found then end for
next
return full URL and file for key

//**** Post Controller
Pass metadata descriptor
Transform metadata descriptor into key
for each vault
     directly access key on vault
     if found then end for
next
if not found then
     locate first writable vault
```

```
        insert file with key
    end if
    return

//**** Lookup Controller
Pass metadata descriptor
Transform metadata descriptor into key
for each vault
        directly access key on vault
        if found then return full URL
next
return not found
```

Turning now to Fig. 3, a process 140 for accessing files stored on one of the vaults 130-132 is shown. The process 140 first generates a key from a metadata file (step 142). The metadata file identifies all elements that make up a single application, as identified using a state probe. The operation of the state probe is described in more detail in a copending application entitled "Automatic Configuration Management," Application Serial No. 08/993,103, filed on December 18, 1997, the content of which is incorporated by reference.

The metadata file describes the elements of the application, including the location of the files and the configuration of the application. The size of the metadata file is typically a small fraction of the size of the total state referenced. The key from the metadata file may be used to access and retrieve component files stored in the vaults 130-132.

If the component files of the application software to be recreated using the key are large and therefore cumbersome to transfer, it is more efficient to determine whether the component files of the application software already exist locally. Such determination may be made by first looking up the key on the client 122 (step 144) and then optionally comparing the key generated from the metadata file to the key on the client 122

~~(step 146) and is detailed in Fig. 3 below. The key comparison process sets a flag if a difference exists and otherwise clears the flags.~~

The difference flag is then checked (step 148). If the difference flag is set, the key generated from the metadata file is used to retrieve software components files from the vault (step 150). Alternatively, if the flag is cleared, the desired file already exists on the client 122 and the process 140 exits (step 152). Pseudo-code for the process 140 is as follows:

```
Transform metadata descriptor into key
Using location attributes of key, locate file on client
Generate key for local file
compare metadata key and local key
if metadata key matches local key then return
for each vault
     directly access key on vault
     if found then retrieve file
next
return success if found
```

Turning now to Fig. 4, the key comparison step 146 of Fig. 3 is illustrated in more detail. First, the difference flag is initialized to zero (step 160). Next, the process 146 determines whether binary data associated with the files being compared are different (step 162). If no difference exists, the process exits (step 176). Alternatively, if the binary data differ, the process then compares the keys based on various sequence attributes associated with each of the files being compared (step 164).

For example, location attributes, defined as directory paths, may be checked. Once the location attributes are determined to be equal, the sequence attributes may be used for further identification. A combination of multiple sequence attributes may be used together to reliably determine identity.

Common examples of sequence attributes may include, but are not limited to, attributes such as date created, date modified, date last accessed, and version number. Certain sequence attributes may take precedence over other sequence attributes. For example, if the version numbers are not equal, date attributes may be ignored.

The process of Fig. 4 checks whether the sequence attributes are equal (step 166). If so, the difference flag is set (step 168). From step 116, if the attributes are not equal, the process checks whether one of the attributes is newer than the other (step 170). If so, the process proceeds to step 168 to set the difference flag. Alternatively, in the event that the attribute is not newer, the process then checks whether or not one of the attributes is older (step 172). If no, the process proceeds to step 168 to set the difference flag. Alternatively, in the event that one of the attributes is older, the process determines whether or not the file may be overwritten (step 174). If so, the difference flag is set (step 168). Alternatively, the process exits (step 176).

Referring now to Fig. 5, a flowchart illustrating a post-process 170 for placing files onto the vault is shown. The post-process 180 first generates a post key from the metadata file (step 182). Optionally, the process 180 may look up the key present on the vault (step 184) and compare the keys (step 186). If the comparison causes the difference flag to be set (step 188), the key from the metadata is used to post the file to the vault from the client (step 190). From step 188 or step 190, the process of Fig. 5 exits (step 192). Pseudo-code for the process 170 is as follows:

```
Transform metadata descriptor into key
for each vault
    directly access key on vault
    if found then end for
next
if not found then
```

locate first writable vault

insert file with key

end if,

return

Turning now to Fig. 6, the process 182 of Fig. 5 to generate the post key from the metadata file is shown in more detail. In Fig. 6, metadata associated with each file is generated (step 194). Next, the process 182 verifies the integrity of the file (step 196).

Integrity is verified using a sufficiently unique byte level check to statistically guarantee that the file is intact. Various known algorithms may be used, including 32-bit checksums, cyclic redundancy checks, and hash-based functions and checksums. While any method which detects a change in the byte ordering of the file may be used, a method with high levels of statistical uniqueness and favorable performance characteristics should be used. For example, MD5 (Ron Rivest, 1992) provides a cryptographically strong checksum.

A key is then generated from the metadata (step 198) before the process 182 exits (step 189). Key generation should include an integrity checksum as described above as well as basic information about the size, name, and attributes of the file. In the form of a checksum, the key allows identity information as well as integrity information to be easily verified.

Turning now to Fig. 7, a process 190 for performing look-ups based on an identity key is shown. The process 190 first enumerates all available vaults in a vault chain (step 192). Next, for each vault, the process 190 generates a universal resource locator (URL) based on the vault and the identity key (step 194). Next, the process 190 checks for the existence of the URL (step 196). If the URL exists in step 198, the vault has been located and the process 190 exits (step 202).

The URL specifies a unique address using one or more protocols to identify and access local and remote resources.

Alternatively, if the current vault fails to offer the correct URL, the next vault is selected (step 200) before the process 190 loops back to step 194 to continue searching all vaults in the vault chain. If all vaults have been searched but the URL is not found, the process returns with an error indication. Pseudo-code for the process 190 is as follows:

```
Transform metadata descriptor into identity key
Construct redundant vault chain
Sort vault chain in order of closest accessibility
for each vault
     form URL based on vault location and identity key
     check existence of URL
     if found then return vault location and URL
next
if not found then return error
```

~~Figs. 8A and 8B show alternate processes to provide~~ state-based software life cycle management using a vault. Turning now to Fig. 8A, a process for performing software management first generates the metadata (step 403), as described in Fig. 1. The metadata may include DNA information, as described in more detail in the incorporated-by-reference application. The information is then used to maintain software (step 405) before the process exits. Correspondingly, Fig. 8B shows a second software life cycle management process. Initially, the metadata information is generated and published (step 410). Next, components of the software are replicated (step 450) based on the metadata. The software is then installed (step 470). After installation, the software may be maintained ~~(step 490).~~

~~Turning now to Fig. 9, the metadata publication step 410 is shown in more detail. In Fig. 9, a vault is located (step 412).~~ The vault may be a server that maintains items referenced in the ~~metadata files. Next, component files associated with the~~

software are stored in the vault (step 414). Similarly, metadata including DNA information is stored in the vault (step 416). A catalog, or an index of metadata files that represent the physical components of the software being published, is updated (step 418). Finally, the process 410 exits (step 420).

Turning now to Fig. 10, the file storing step 414 of Fig. 9 is shown in more detail. Each item in the metadata file is initially selected (step 432). The integrity of the item is verified (step 434). The metadata information is used to verify the integrity of the item. Simple sequence and identity attributes are compared to ensure that no change has occurred to the file. The fastest comparisons are performed first with slower but more reliable checks being performed later. By using a combination of attributes which may include the date accessed, date modified, version number, date created, multiple file checksums, block checksums, complete byte comparisons, secure file hashing, and file attribute comparison, the integrity of the file may be reliably correlated to the information in the metadata.

Next, a unique identification value is generated (step 436) based on the location of the file in the vault, and a search for a copy of the file is done to determine the existence of the file (step 438). If the copy of the file does not exist in the vault, then the file is copied into the vault (step 440). From step 438 or step 440, the process of Fig. 10 determines whether additional items remain to be processed (step 442). If so, the process of Fig. 10 loops back to step 432 to continue processing the next item. Alternatively, the process exits (step 446).

The replicate step 450 of Fig. 8B is shown in more detail in Fig. 11. First, the source vault is located (step 452). Next, the destination vault is located (step 454). Files are then transferred from the source vault to the destination vault (step 546). Similarly, metadata information is copied from the source vault to the destination vault (step 458). Finally, the vault catalog is updated (step 460) before the process of Fig. 11 exits

(step 462).

Turning now to Fig. 12, the installation step 470 (Fig. 8B) is shown in more detail in Fig. 12. First, the vault catalog is loaded (step 472). Next, the highest version of the software stored in the vault is determined (step 474). The metadata associated with the highest version of the software is copied to the target machine (step 476). Further, data is remapped (step 478). The process of Fig. 12 then applies a preprocessing operation (pre-RNA) to the remapped data (step 480) to convert data into the proper format and set up variables appropriately, among others. Further, items associated with the software are installed (step 482). A post-processing (post-RNA) process is applied (step 484). This step is similar to step 480 in that variables are checked and data is formatted. Finally, an inventory of the software being installed is updated (step 486) before the process exits (step 488).

Turning now to Fig. 13, the maintenance step 490 of Fig. 8B is shown in detail. The maintenance step 490 is an event driven process and thus receives a software trigger event (step 492). Based on the trigger event, the process of Fig. 13 determines various possible events, including a check for update event 494, a protect software event (step 495), a software recovery (step 498) event, a check removal event (step 500), and an examine system event (step 502). From steps 494-502, the triggering event is reported (step 504) before the process of Fig. 13 exits (step 506).

In the manner discussed above, the vault can inventory, install, deploy, maintain, repair and optimize software across LANs and WANs. The vault installs only known working software combinations and insures against deletion of shared components, thus protecting user investments in existing solutions for enterprise-wide systems management, network management, and application management.

The techniques described here may be implemented in hardware or software, or a combination of the two. Preferably, the

techniques are implemented in computer programs executing on programmable computers that each includes a processor, a storage medium readable by the processor (including volatile and nonvolatile memory and/or storage elements), and suitable input and output devices. Program code is applied to data entered using an input device to perform the functions described and to generate output information. The output information is applied to one or more output devices.

Each program is preferably implemented in a high level procedural or object-oriented programming language to communicate with a computer system. However, the programs can be implemented in assembly or machine language, if desired. In any case, the language may be a compiled or interpreted language.

Each such computer program is preferably stored on a storage medium or device (e.g., CD-ROM, hard disk or magnetic diskette) that is readable by a general or special purpose programmable computer for configuring and operating the computer when the storage medium or device is read by the computer to perform the procedures described. The system also may be implemented as a computer-readable storage medium, configured with a computer program, where the storage medium so configured causes a computer to operate in a specific and predefined manner.

While the invention has been shown and described with reference to an embodiment thereof, those skilled in the art will understand that the above and other changes in form and detail may be made without departing from the spirit and scope of the following claims.

What is claimed is: